Lab5 – File IO, random numbers, timing code, finding the **mode**, "Big Oh", and refactoring

(0) Download the Starter Kit from moodle (onCourse). **Name: _____**

(1) We'll do a quick **walkthrough** of the code at the start.

(2) Input/output can be difficult in C++. This is also true in Java *IMHO*. Rather than fumble around at the start, I've given you a function called **`makeRandomDataFile()`** to (i) create a new file (as entered by the user via `stdin` [*hey, what is stdin?* _____]; (ii) write to that file with random numbers where the exact count of random values is also entered via `stdin`; (iii) Note: that the maximum number of values you're fixed size array will handle is set in a constant in main() (see `main.cpp`), as is the upper bound of the range of random values you'll generate, e.g., **1000** random values in the range from 1 to **10**.

Answer these Questions about **`makeRandomDataFile()`**:

(a) Why did I have to use "**`c_str()`**" here: `outFile.c_str()` ?

(b) What case(s) would cause this statement to be true?

```
if ( fout.fail() )
```

(c) *Find* your new output file (where *is it?*); what is the **pathname** for this file?

(3) Once you have answered these three questions and you have successfully made a file filled with random numbers, **call us over**.

(4) Now you write a function **`readDataFile()`** in IO.cpp to read *in* the data from your new file of random values and load that data into an array called Data. The argument "`hmr`" stands for "**h**ow **m**any *(values)* **r**eally" are in the array. In C/C++, you always need an extra variable with your arrays (static or dynamically allocated) since your arrays will not always be filled to capacity, thus you need to know *how many* values are *really* in your array.

```
void readDataFile (string filename, long Data[], long& hmr);
```

(5) How do you *know* that the data values are *really* stored in your array, Data? How can you "prove" it. Call us over when can verify that your values *are in the array.*

(6) Find the **mode**[1] of your array of random values in two ways:

(a) **`findMode_1()`**: **Sort** the values in the array. Then walk through the array to find the number that has the longest "run" of values. For example: 1,1,3,4,4,4,4,4,4,5,5 has a mode of **4** since the value 4 appears more times than any other number. As you can see, I have written this one for you. Answer some questions about this code.

(i) We are using the built-in function **`qsort()`** to sort an array of values. Read up on `qsort()`; explain the arguments (no hand-waving).

```
// (1) sort the data
qsort(Data, hmr, sizeof(long), compareMyType);
```

Data:
hmr:
sizeof(long):
compareMyType:

(ii) What is happening between `qsort()` and `compareMyType()`? Where does `compareMyType()` get called?

(iii) You finish the algorithm for this solution.

(iv) Using the timer code, how long does it take you to find the mode using this method on one million random numbers in the range from [1..10], inclusive?

(v) Assume your value of hmr is N. If the `qsort()` algorithm, on average, runs in `O(N*lg(N))` time, what is the overall Big Oh for the **`findMode_1()`** algorithm?

(b) **`findMode_2()`**: "**Refactor**" your algorithm from findMode_1 by using a different algorithm. Rather than sort, use a Histogram array, that is, use another array to indicate the count of the number of times each value from 1 to 10 appears.

(i) Using the timer code, how long does it take you to find the mode using this method on one million random numbers in the range from [1..10], inclusive?

(ii) Assume your value of hmr is N. What is the overall Big Oh for the **`findMode_2()`** algorithm?

[1] You can assume that you need only report one mode if you have a multimodal situation.